Electronic Notes in Theoretical Computer Science 30 No. 4 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume30.html 21 pages

Debugging Prolog Using Annotations

M. Kulaš

FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract

We present an annotation language well-suited for rendering aspects of Prolog execution. Our annotations are special Prolog goals that act as executable comments, performing debugging at run-time. No restrictions are placed upon the object language, the concern being verification of (full) Standard Prolog programs. Here we discuss the merits of the annotations for Prolog debugging. All the examples are actual runs of our system, *Nope*.

1 Introduction

We start by briefly surveying previous research in debugging of Prolog programs. Then we summarize our annotation language, introduced in [7]. Sec. 4 and Sec. 5 offer a close look into the potentials of such a language for debugging Prolog, by means of extensive examples. In particular, Sec. 5 shows how easily one obtains quite a useful declarative diagnoser out of two annotations. Sec. 6 rounds off the survey of related work.

2 Debugging Prolog

In the area of Prolog debugging and validation different approaches have been put forward.

The very influential approach of declarative debugging pioneered by [14] has an inherent limitation of assuming declarativeness of the debugged predicates. Namely, the predicates have to be instantiation-stable ('monotone') in the sense that the properties of success/failure hold under substitutions: if a goal succeeds/fails, then any instance of it shall succeed/fail as well. This clearly does not have to hold if `var/1` or `nonvar/1` is allowed, which is the case in full Prolog.

Another very interesting declarative approach, LPTP [15], proposes a first-order language for rendering procedural properties of Prolog (success, failure and universal termination of goals) in a declarative way, together with a powerful calculus, formalizing a suitable execution model, for interactively proving these properties. The inherent limitation here is akin to the previous approach: only a logical subset of Prolog (no cut, var or assert/retract allowed) can be treated.

The traditional verification paradigm of pre/post conditions has first been adopted for full Prolog in [4] and further promoted especially by [6], in the generic framework CIAO for program development and debugging. In addition to pre/post conditions, their annotation language offers the so-called comp annotations, like in :- comp qsort(A,B) : (list(A), var(B)) + does_not_fail, expressing some properties of the computation (here: non-failure of the query qsort(A,B)). Behind the somewhat patchworky syntax is a more serious issue of expressive power: pre/post conditions alone do not suffice to express every property of execution, e. g. the non-failure. As stated in [6], *"no property which refers to (a sequence of) intermediate states in the computation (...) can be easily expressed using calls and success assertions only".* This shortcoming of the pre/post conditions is here being compensated for by handing the more tricky properties over to Prolog.

For the benefit of the user, as well as for automation of debugging, it is advantageous to start from an explicit execution model. The model should ideally be simple and complete.

One such model happens to be generally known. The 4-port model [2], which is the basis of the standard Prolog debugger, is a *complete* execution model of Prolog, in the sense of entailing every aspect of Prolog execution of necessity for verification, and therefore all the information one might need. This model maps a query Q to its *standard trace* $\mathcal{T}(\mathsf{Q})$, which is a sequence of *events* of the form Port $\mathsf{F}(\mathsf{T}_1, .., \mathsf{T}_\mathsf{N})$, where Port may be one of {call, exit, fail, redo}, F/N are predicates, T_i are terms, representing the atomary steps of the Prolog interpreter during the execution of the query Q (as defined in [2,16]).

A consequence of such expressive power is also that the amount of information generated by this model be overwhelming, so that its straightforward use as in the standard 4-port debugger is often considered as too low level.

One line of research addressing this problem is *trace analysis* of [5], and it advocates building (essentially) the standard trace and filtering the non-interesting events out of it. The approach bases upon primitives for sequential forward and backward traversal of the standard trace, which can be aggregated into arbitrarily high-level filters. The user starts from an erroneous query Q and interactively conceives and discards hypotheses about the culprit by selecting and assessing different subsets of the standard trace $\mathcal{T}(\mathsf{Q})$. These subsets can be arbitrary ('abstract views of the execution'), and are built by filtering out. Starting from one subset the user can specialize it, or generalize, or replace altogether, until reaching the end of the trace. This performs best as exhaustive checking with dynamically changing focus.

An alternative line of research based on the 4-port model is pioneered by

[11], and it advocates tracing only the interesting events. The user starts typically from some hypotheses and hopes to hit upon any queries that might violate them. In some cases the hypotheses can be statically proved as valid. We call this paradigm *event transforming* because of its typical implementation. As opposed to a trace-analysis system like Opium [5], where the execution of the user's program Π will be sampled at all ports of all predicates, in a transformation system like Advice [11] only the execution of some selected goals will be re-routed, so virtually, not the user's original program Π but a *transformed* program Π' will be executed.

Our own approach [7], realized in the system Nope, is event transforming.

None of the two paradigms (trace analysis vs. event transforming) is a special case of the other. In a transformation approach, one can as well start from an erroneous query `Q` and use trivial 'invariants' that only serve to construct a selected subset of $\mathcal{T}(\mathtt{Q})$, hereby emulating trace analysis. However, here one cannot change the subset mid-query and is stuck with it, for better or for worse, till the end of the query execution. Analogously, one could emulate event transforming within trace analysis by using the invariant as the filter. For example, at the call port of `\+ Q` filter out everything which is ground. If this produces a non-empty set, then the invariant is violated. However, this is clearly far more inefficient than using a dedicated transformation (checking only `\+ Q`), due to the extreme waste of overhead. Apart from this, in case of several simultaneously present invariants there would probably have to be several trace traversals [5], as opposed to only one in the other paradigm, since in general it isn't clear in advance which event is going to happen first (so the first event may never happen, which is bad luck for the second one).

As a rule of thumb one might say that trace analysis is better suited for dynamic checking of expensive queries, where as much information as possible shall be tapped in order to be rolled back and forth many times using different abstract views. One of its main assets lies in the completeness of the standard trace, permitting exhaustive exploration of the bugs, especially combined with trace recording. Further, freely abstracting over the sequence of events, compare the quote from [6] on page 2. Finally, the ability to modify the checking 'on the fly' (during the execution of the query): depending on the outcome of the current filter, a new one may be used in the sequel. Event transformers cannot modify their checks mid-query.

On the other hand, event transformation, having a very low runtime-overhead and allowing for quite useful annotation languages (as we hope to show in the following), is better suited for quick prototyping and monitoring. The invariants serve also as valuable program documentation. Some classes of them can be checked statically.

3 Nope annotation language (summary)

To make this article self-contained, let us recall the language of [7].

Annotations describe properties of predicates. Such properties can be pre or post conditions, which must hold true when a predicate is called or exited, respectively. Our concept transcends pre/post conditions: we introduce two more kinds of annotations, fail and redo annotations, hence incorporating a whole model of Prolog execution into our language. This enables natural rendering of many procedural properties of Prolog which cannot be expressed with only pre/post conditions, like non-failure. There are four more novelties in our approach. First, any annotation can be "narrowed down" to a subset of calls, via templates and contexts, giving much more flexible assertions. Notably the novel idea of calling context adds significant expressive power, giving access to arbitrary program points. The annotations are defined simply as Prolog goals, making them fully parametric and therefore very comfortable for debugging. Finally, the annotations are applied via a general kind of matching instead of unification, enabling the use of local variables.

3.1 Call annotations

Motivated by the need to express more precisely the intended calling patterns of a predicate, we introduced in [7] call annotations, with the following syntactic variants.

$$\begin{aligned} \mathsf{call\ AtomicGoal} &\Rightarrow \mathsf{MustHold} \\ \mathsf{call\ AtomicGoal\ with\ Template} &\Rightarrow \mathsf{MustHold} \\ \mathsf{call\ AtomicGoal\ \{\ with\ Template\ \}\ within\ Context} &\Rightarrow \mathsf{MustHold} \end{aligned}$$

An example for the first variant is `call append(X,Y) => nonvar(X); Y=[]`, detailed on page 12. There will be diverse examples in this paper, each tested with Nope. Nope is an (almost standard) Prolog module that recognizes our annotations and checks them at run-time.

The second variant is an enhancement of the goal specification in Prolog, namely a template allows us to specify the calling pattern more precisely than with head unifications alone. In fact, arbitrarily precise, since a template can be any Prolog goal. The default template is `true`.

The third variant introduces the idea of *calling context*, a way of specifying sets of program points. The contexts are partially ordered (*context subsumption*). The default context is `_`, which subsumes any other context, so an annotation without a context part applies in any context.

In case of annotation failure, Nope will issue a warning. The default warning in Nope shows the violated annotation, with its arrow broken, like this

$$\mathsf{/*\ NOPE\ call\ append(A,[1,2])} \not\Rightarrow \mathsf{nonvar(A);\ [1,2]=[]\ */}$$

This can be overridden via the `else/2` predicate (which is defined on page 10): `call append(X,Y) => nonvar(X); Y=[] else write(query_diverges)` customizes the warning into `/* NOPE query_diverges */`.

Definition 3.1 (call annotation) *Let* P *be a predicate of arity* n*. A call annotation for* $\mathsf{P/n}$ *is a Prolog goal*

$$\mathit{call\ Premiss} \Rightarrow \mathit{Constraint}$$

where Premiss ::= Goal { with Template } { within Context }*. Here* Goal *is a Prolog goal for* P/n*, i. e. a Prolog term of the form* $P(S_1, .., S_n)$*.* Context *is a context (see Sec. 3.1.2).* Template, Constraint *are arbitrary Prolog goals.*

Technical note 1: We show our annotations mostly in the traditional, declaration form :- Annotation. to make them stick out, but please note that they are simply *goals* and can be called like any other Prolog goals, amounting to *parametric* annotations. (In case you wonder which predicate is here being called, it's.. right, the arrow ⇒/2.)

Technical note 2: Our annotation language is implemented by means of *goal replacement*. Therefore, in principle there is nothing to prevent us from annotating *arbitrary goals* and not just *'atomary'* ones (as opposed to composite predications like conjunction, disjunction, if-then-else or negation). For example, it is just a question of slowing down the parser a notch to allow arbitrary negated goals, making it possible to express safe negation in its most general form (see Sec. 4.2). The default parser of our implementation will recognize annotated negation, but conjunction, disjunction and if-then-else will be transparent, i. e. any annotations for these built-in 'predicates' of Standard Prolog will be ignored, as opposed to all other built-ins (and of course all user-defined predicates).

We shall assume every call annotation in the form call Goal with Template within Context ⇒ Constraint, which we may do owing to the defaults for Template and Context.

The annotations shall not be applied via unification, since we consider matching (subsumption) to be the more natural choice. Recall that Pattern subsumes Object if Object is an instance of Pattern, i. e. if Object can be unified with Pattern without further instantiating any variables in Object. We generalize this so as to accommodate not only equality but arbitrary goals in Template: everything goes but there is an invariant, the user's goal, which may not be changed.

Definition 3.2 (succeeds soft) *Let* Goal *and* Invariant *be arbitrary Prolog goals. We say that* Goal *succeeds soft on* Invariant *if* Goal *succeeds without changing* Invariant*.*

More formally: If Goal succeeds with an answer substitution σ, then σ may only rename Invariant, written as $\sigma(\mathsf{Invariant}) \cong \mathsf{Invariant}$.

Soft succeeding goals are not only of the kind \+ \+ G. Nope provides a predicate call_soft(G, J) that succeeds only softly on J for any G and any J. The advantage over \+ \+ G is that call_soft allows for *local variables*, because the variables of G not occurring in J may get instantiated.

In Def. 3.7 we define applicability of an annotation via soft success. Owing to this we may have arbitrary premisses, like p(X,Y) with X=Y.

3.1.1 Simple context

It is useful to be able to access the immediate ancestor (parent) of Q, i. e. the goal which called Q.

Definition 3.3 (parent of a goal) *The head of a clause is the parent of each body goal. If a goal is issued at the top-level of a Prolog interpreter, we will say that its parent is the goal* false*.*

To capture the notion of the parent, we introduce *contexts* in the annotations.

A context shall represent the parent of the annotated goal. In the simplest case, it can be any atomary Prolog goal. So the meaning of an annotation for Q within Context shall be to check on Q occurring anywhere in the clause bodies of Context. One can view contexts as specifying *sets of program points*, namely those where the annotated goal might occur (in this first simple case: all points in the bodies of Context).

Furthermore, sometimes it is useful to exclude contexts. For example, if we only want to check the non-recursive calls of a predicate P, we need to express "check P within any context but P". For such purposes we allow *negative contexts*, represented as dash-prefixed atomary Prolog goals. The meaning of -Context is the complement of the set of program points represented by Context, i. e. this can be any point in the program (plus top-level, 'defining' false/0) outside of the definition of Context.

In the following we shall further refine the idea of a context. But first we have some questions for our contexts so far. One key question about contexts is their inheritance, i. e. when is an annotation for Q within X pertinent to Q within Y? Also, when is an annotation bound to a context CA pertinent to a goal within a context CQ? To settle these questions, we define a partial ordering for contexts.

We start from the natural lattice of first-order atomary formulas (augmented by a 'universal' and a 'null' formulas) modulo renaming, as introduced by [13]. The partial ordering $\preceq$ being "is instance of", the greatest lower bound of two atoms being their most general unifier (or the special bottom element *null formula*) and the least upper bound being their most specific generalizer (or the special top element *universal formula*).

Next we enhance the language. Our language is the following set $\mathcal{C}$ of atomary and non-atomary formulas (plus the universal and the null formula): the first-order language – modulo renaming – made of standard Prolog variables, function and predicate symbols, and a unary operator -. Atoms Q of this language we call *positive contexts*, and non-atoms -Q we call *negative (negated) contexts*. The universal formula is represented here by _, and the null formula is represented by -_.

It remains to enhance the partial ordering.

Definition 3.4 (partial ordering of contexts) *For positive contexts* X *and* Y *we define:* $\text{-}X \preceq \_$, $X \preceq Y$ *if* X *is an instance of* Y, $X \prec \text{-}Y$ *if* X *is not unifiable with* Y, $\text{-}X \preceq \text{-}Y$ *if* $Y \preceq X$.

If we assign to every atom Q from $\mathcal{C}$ the set of its ground instances, and to -Q all the other ground atoms in the Herbrand base $B_{\mathcal{C}}$ for $\mathcal{C}$, we obtain an isomorphism between $(\mathcal{C}, \preceq)$ and a sub-poset of $(\mathcal{P}(B_{\mathcal{C}}), \subseteq)$. Due to this isomorphism, $(\mathcal{C}, \preceq)$ is a poset and the seeming asymmetry of the definition actually is a perfect duality: $X \prec \text{-}Y$ iff $Y \prec \text{-}X$ iff X,Y are not unifiable.

Definition 3.5 (context subsumes) *If* $CQ \preceq CA$*, then we say: the context* CA *subsumes the context* CQ*.*

We chose the names CA and CQ to suggest *annotation context* as opposed to *query context*. The interesting case is namely when an annotation context subsumes the actual query context, meaning the annotation may fire.

Definition 3.6 (context inheritance) *If* $X \prec Y$*, then an annotation for* Q *within* Y *shall be inherited to* X*, i. e. it is also an annotation for* Q *within* X*.*

In examples of Sec. 4.3 and Sec. 5 we will advocate usefulness of contexts.

3.1.2 (Generalized) Context

Assertions can be divided into predicate assertions, stating properties of a predicate, and program-point assertions, stating properties that shall hold when the code at a given point in the program is about to be executed. Traditionally, the two classes have different syntax, predicate assertions being written as declarations, and program-point assertions being scattered all over the program code (like in [6]). Our annotations are predicate assertions. However, due to contexts, as defined so far, we can capture some sets of program points. For illustrations see Sec. 4.3.

There is only a small step necessary to be able to capture also individual program points. For example, say we have a program-point annotation $\mathcal{A}$ placed as follows:

$$\mathsf{H :\!- B_1, .., B_i,\ \mathcal{A},\ B_{i+1}, .., B_n.} \qquad \textit{\% k-th clause for the predicate of H}$$

then we can simulate it in Nope as

$$\mathsf{call\ B_{i+1}\ within\ H{*}pp(k,i)} \Rightarrow \mathcal{A}$$

In other words, we enhance the context specification by an optional part, program-point, linked to the simple context by an asterisk. A non-asterisk term Context is regarded as an abbreviation of Context * _ meaning "any program point within the simple context Context". The program-point is a Datalog term representing the intended program point(s) like pp(K,J), where K is the serial number of the clause in the definition of the parent goal, and J is the serial number of the body subgoal (counting each conjunct/disjunct/implicationpart). The partial ordering $\preceq$ of simple contexts shall be lifted in the obvious way to generalized contexts.

3.1.3 Calling premiss

For the premiss of an annotation we will also use the more evocative name *calling premiss.*

Definition 3.7 (calling premiss subsumes goal) *We say that a calling premiss* Goal with Template within Context *subsumes the call port of* Q *(for short: subsumes* Q*), if at that port* Q=Goal, Template *succeeds soft on* Q*, and also* Context *subsumes the parent of* Q*.*

More formally: Let Parent be the parent goal of Q. The following three conditions must hold. If σ is the answer substitution for $\mathsf{Q{=}Goal}$, then $\sigma(\mathsf{Q}) \cong \mathsf{Q}$. If θ is an answer for $\sigma(\mathsf{Template})$, then $\theta(\mathsf{Q}) \cong \mathsf{Q}$. Lastly, $\sigma(\mathsf{Parent}) \preceq \sigma(\mathsf{Context})$.

A logical reading of this definition (leaving out the contexts) shall be:

$$\forall\, \mathcal{V}(\mathsf{Q})\ \exists\, \mathcal{V}(\mathsf{Template}) \backslash \mathcal{V}(\mathsf{Q}) \quad \mathsf{X}_1 = \mathsf{Q}_1, .., \mathsf{X}_n = \mathsf{Q}_n \vdash \mathsf{X}_1 = \mathsf{G}_1, .., \mathsf{X}_n = \mathsf{G}_n,\ \mathsf{Template}$$

Here $\mathsf{Q}_1, .., \mathsf{Q}_n$ are respectively the $\mathsf{first}, .., \mathsf{nth}$ arguments of Q, i. e. the actual calling substitution, and similarly $\mathsf{G}_1, .., \mathsf{G}_n$ are the arguments of Goal, i. e. the calling substitution monitored by the annotation. $\mathsf{X}_1, .., \mathsf{X}_n$ are new variables. The monitored calling substitution together with $\mathsf{Template}$ constitutes the full (i. e. normalized) monitored calling template. $\mathcal{V}(\mathsf{Term})$ is the set of variables in Term. So the actual calling substitution shall imply the monitored calling template.

Observe that this allows local variables, namely variables that don't occur in Goal can be used to create and propagate values from the calling premiss throught to the constraint.

Now we can define what does it mean to apply a call annotation.

Definition 3.8 (applicable) *An annotation is applicable on a goal* Q *if its calling premiss subsumes* Q*.*

Definition 3.9 (call annotation satisfied/violated) *An applicable call annotation is satisfied/violated for a goal* Q*, if its constraint succeeds/fails at the call port of* Q*.*

Definition 3.10 (correctness wrt annotations and queries) *An annotated program is said to be correct if, during the computation of arbitrary queries (from the given set of allowed queries), all the applicable annotations are satisfied.*

3.2 Following the execution model

In the previous subsection we suggested annotating the call port of a goal, in order to grasp more precisely the intended calling patterns. So what about the other three ports? Here we refer to the classic model of Prolog execution [2], abstracting a goal to a black box with four ports, known from the debugging sessions.

Following this model, we propose a language of annotations as a step towards capturing procedural aspects of Prolog. This language has the advantage of there being a workable consent among Prolog users on what the box trace model of Prolog execution means. We consider it a head-start for an annotation language to be based upon standard concepts.

Definition 3.11 (port annotation) *Let* P *be a predicate of arity* n. *A* Port *annotation for* $\mathsf{P/n}$ *is a Prolog goal of the form*

$$\mathit{Event} \Rightarrow \mathit{Constraint}$$

where

Event	::=	*Port Premiss*
Port	::=	*call | exit | fail | redo*
Premiss	::=	*Goal { with Template } { within Context }*

Here Goal *is a Prolog goal for* P/n, *i. e. a Prolog term of the form* $P(S_1, .., S_n)$. Context *is a context.* Template, Constraint *are arbitrary Prolog goals.*

As in [4], we deal only with *partial correctness*, because we do not claim whether goals actually succeed, fail or redo. Or even get called. Also, as the transformational semantics below reveals, our annotation language provides for accessing the values of variables on call and on exit from a query, similar to [4]. Thus, our language is also a language of *binary assertions*. The actual calling pattern is accessible on the left of $\Rightarrow$, and, in case of exit and redo annotations, the exit pattern is on the right. This circumvents the need for language primitives to access the call/exit values.

Applicability of a port annotation is defined as in Def. 3.8. To define the meaning of port annotations, we can adopt Def. 3.9. But observe that a premiss shall always match the *call* port of the goal (therefore we dubbed it 'calling premiss').

In this paper we do not define the concept of a port. But still, we do give a precise semantics of our annotations, via a program transformation. Namely, we show the essentials of our implementation of the run-time checking of annotations. The implementation bases upon the following program transformation. Each annotation for a predicate P imposes a virtual transformation on the user program: each goal $\mathsf{P}(\mathsf{T}_1, .., \mathsf{T}_n)$, be it a top-level query or in a body of a clause, will be computed as if changed according to the following table (here we abstract from contexts and bookkeeping to obtain a simpler picture). The labels on the arrows indicate the respective Port of the annotation.

Note that a call annotation hence corresponds to a (global) precondition, and an exit annotation to a (global) postcondition. The usual coupling is achieved via an own (local) precondition to a postcondition.

Lemma 3.12 (non-interference of port annotations) *Port annotations*

$P(T_1,..,T_n) \overset{\text{CALL}}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$ (Constraint else Warning), $P(T_1,..,T_n)$
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{\text{FAIL}}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$ ($P(T_1,..,T_n)$; (Constraint else Warning), fail)
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{\text{EXIT}}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$ $P(T_1,..,T_n)$, (Constraint else Warning)
; $P(T_1,..,T_n)$

$P(T_1,..,T_n) \overset{\text{REDO}}{\rightsquigarrow}$
call_soft(FullTemplate, $P(T_1,..,T_n)$) $\rightarrow$ $P(T_1,..,T_n)$, (true; (Constraint else Warning), fail)
; $P(T_1,..,T_n)$

Fig. 1. The program transformation of Nope

in Nope are not interfering with the success set: A query about the user's program Π *computes the same answers and in the same sequence when posed to* $\Pi+\Delta$*, where* Δ *denotes arbitrary port annotations with terminating and pure templates and constraints.*

Annotations with a non-terminating template or constraint are clearly a bad idea. But otherwise, this lemma imposes the too strong restriction of purity upon templates and constraints. In practice, users profit from annotations with side-effects. Like [6], we guarantee that no amount of annotation tweaking will change any variables of a user's query. As for the rest of dangers, we simply count on users to be sensible in their choice of side-effects. Lemma 3.12 follows from the program transformation of Nope (Fig. 1), where call_soft/2 and else/2 have the following definitions:

```
call_soft(Test, Invariant) :-
  copy_term(Invariant, InvariantCopy),
  once(Test), % Test succeeds, but is it harmless on Invariant?
  variant(Invariant, InvariantCopy). % Other variables may be propagated.

else(Constraint, Warning) :- % Constraint else Warning
  \+ Constraint -> (\+ Warning -> true; true)
  true.
```

Here variant/2 checks whether its arguments are renamings of each other.

Lemma 3.13 (compositionality of port annotations) *Port annotations in Nope are compositional: If* $\mathit{Premiss}_k$ *of a stated (i. e. computed) annotation* A_k *subsumes the call port of* Q*, then* $\mathit{Constraint}_k$ *is going to be computed at the* Port_k *of* Q*, no matter how many annotations, for what ports and in what order, were stated until the call.*

This lemma follows from the program transformation of Nope.

4 Illustrations

Here we give several short illustrations in support of our claim that the port annotations are a versatile and an intuitively appealing means for debugging Prolog, enabling the user to specify general mode information, to express and support the verification of hypothetical invariants, to write customized tracers, to alert to unwanted events, and generally enhancing the prototyping and self-reflective capacities of Prolog.

4.1 Mergesort and its partial specification

As a first illustration of our debugging concept, we show a buggy implementation of the predicate merge/3, which expects as inputs two sorted lists of integers and merges them into a sorted output list.

```
:- ensure_loaded(library(nope)).  % how to activate NOPE

% Precondition: the inputs should make sense
:- call merge(X, Y, Z) => sorted(X), sorted(Y)
      else format('inputs ~w and ~w must be sorted', [X,Y]).

% Postcondition
:- exit merge(X, Y, Z) with sorted(X), sorted(Y) => sorted(Z).

merge([X|Xs], [Y|Ys], [X|Zs]) :- X>Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,X|Zs]) :- X =:= Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X<Y, merge([X|Xs], Ys, Zs).
merge([], [X|Xs], [X|Xs]).
merge(Xs, [], Xs).

sorted([]). sorted([_]). sorted([X,Y|L]) :- X =< Y, sorted([Y|L]).
```

After loading this program, we can have the following interaction

```
| ?- merge([3,1], [], [3,1]).
/* NOPE inputs [3,1] and [] must be sorted */

yes
| ?- merge([1], [2], M).
/* NOPE exit merge([1],[2],[2,1])with sorted([1]),sorted([2]) ⇏ sorted([2,1]) */

M = [2,1] ?
```

The second message is warning us that the postcondition is violated, namely the merging algorithm produces unsorted lists (the first and the third clause for merge/3 indeed have swapped comparisons).

An avid reader might object that we do not need the template sorted(X), sorted(Y) in the postcondition, since this is already being taken care of by the precondition. Well this is not quite true. Note that there can be arbitrarily many annotations for a predicate, even for the same port of it, and they will be composed via conjunction. But still, each annotation is regarded as an independent entity, in the sense of not having to take into account any other annotations. So our exit annotations actually do not assume that any call annotations have to be satisfied. For example, if we omit the template above and prove the following two annotations to hold

```
:- call merge(X,Y,Z) => sorted(X), sorted(Y).
:- exit merge(X,Y,Z) => sorted(Z).
```

with respect to the given program and a closed set of top-level queries, then we know that merge/3 is only going to be called with sorted inputs, and that the output is also going to be sorted. But if the call annotation fails for a certain goal, the exit annotation is still going to be checked in Nope, requiring too much of the predicate (namely, that merge/3 is always going to deliver a sorted output, regardless of inputs), compared to its definition.

So the call annotations are 'global' preconditions, not bound to any postcondition. On the other hand, we do have a way of *binding* a certain precondition to a certain postcondition: via the premiss of an exit annotation.

4.2 Some modes

To express that \+ does not alter its argument, we can use the following mode declaration

```
exit \+ X with copy_term(X,X0) => variant(X,X0)
```

This example shows how to access the call/exit values of variables.
Safe negation in its most general form can be expressed as

```
call \+ X => ground(X)
```

The constraint that the first argument of append/2 from [12] be a *complete list*[1] can be expressed (and documented) through the following two annotations.

```
:- call append(X,Y) with var(X), \+ Y=[] => false else write(divergence).
:- call append(X,Y) with var(X), \+ \+ Y=[] => false else write(trivial_answers).

% append(-, _) :- !, fail. % reject incomplete lists  %% now catched by the ann's
append([], []).
append([L|Ls], List0) :-
  append(L, List1, List0), append(Ls, List1).
```

Note that, by using the default context, we do not need to actually check the list completeness, but only the var property, since each recursive call of append/2 will be checked. Note also how the constraint false serves to alert, at runtime, to unwanted events like ill-defined modes.

[1] A *complete list* is a dot-structure with the ultimate tail [ ], as opposed to an *incomplete list*, whose ultimate tail is a variable.

4.3 Use of contexts and parametricity

Due to parametricity and contexts, our annotations can be seen as macro definitions for program-point annotations. For example,

```
lr_alert(Q) :-
    functor(Q,F,N), functor(Q0,F,N),
    (call Q within Q0 => \+ subsumes_noc(Q,Q0) else bark, write(lr:Q)).

subsumes_noc(X,Y) :- groundcopy(Y,Yg), X=Yg.

:- is_my_predicate(X), lr_alert(X), fail; true.
```

monitors leftmost (assuming left-to-right execution) recursive calls of all my predicates, like this

```
| ?- append(X,[Y]).
/* NOPE lr:append(_347,[_94]) */
Prolog interruption (h for help)? c
/* NOPE lr:append(_758,[_94]) */
Prolog interruption (h for help)? a
Execution aborted
```

The utility `bark/0` tunes the warning handler of Nope to interrupt Prolog after one warning. The warning handler provides uniform and graded (and fully customizable) means of handling special events like 'unkosher' modes.

4.4 Uniform conditioning

As a hint on the merits of conditioning even the call annotations (in order to express dependencies between the input arguments), see

```
:- call functor(T,F,N) with var(T) => nonvar(F), nonvar(N).
```

Observe that here we can get rid of the template, via a less readable but equivalent

```
:- call functor(T,F,N) => var(T) -> nonvar(F), nonvar(N); true.
```

However, this does not hold in general. Namely, the *call* values of variables are normally not identical with their exit values. Thus, templates are not redundant, even in case of pure logic programs. [2] For programs with side-effects just more so, because (in case of exit, fail and redo) on the left of the arrow we have the state *before*, and on the right of the arrow: the state *after* the computation of Q.

[2] Another reason is the subsumption on the left vs unification on the right of the arrow, but that is not a vital design decision of the language, the way the call/exit values are.

4.5 Tracing

The Byrd tracer is a special case of annotation composition. We don't have to bear with all four ports of a general goal pattern if we only want a specific port of a specific goal pattern traced.

```
fail_alert(X) :- fail X => write('Fail: '), write(X), nl.
:- fail_alert(p(X)).

p(1) :- p(2), p(3).
p(2) :- p(4).
p(4).

| ?- p(K).
Fail: p(3)
Fail: p(4)
Fail: p(2)

K = 2 ?
```

4.6 Finite failure

The fail-port is capturing 'no (more)' rather than 'no'. Often we are interested only in genuine 'no', the finite failure. We can derive the *whole* of this new event from the basic four events, for example by emulating the Prolog stack:

```
ff(( Q with T within Ctx => C )) :-
(call Q with T within Ctx => asserta(lemma(Q:call))),
(redo Q with T within Ctx => asserta(lemma(Q:redo))),
(exit Q with T within Ctx => once(retract(lemma(_:_)))),
(fail Q with T within Ctx => once(retract(lemma(Q:What))), What=call -> C; true).
```

Alternatively, we can provide finite failure as a regular annotation, say ff, on a par with call/exit/fail/redo. This is a matter of only adding a new entry in the transformation table of Nope, amounting to two new clauses, and allows for superior code. Either way, we may now write

```
ff_alert(X) :- ff X => write(finitely_fails:X), nl.
:- ff_alert(p(_)).
```

and obtain the desired behaviour:

```
| ?- p(K).
finitely_fails: p(3)

K = 2 ?
```

Non-failure, which is not expressible using pre/post-conditions alone, can now be expressed as

```
:- ff Premiss => false.
```

So the subset assertions of [6] or positive examples of [9] would here look like

```
:- ff qsort([2,1],[1,2]) => false.
```

Similarly to finite failure, we can *derive* the event of building a resolvent (the unify port), or *build* it *in*.

5 Toy declarative diagnoser

It is curious to note that quite a useful (and efficient) diagnosing heuristics can be made of two simple parametric annotations, plus two standard utilities like query-the-user and demo.

The idea is to designate a set $\mathcal{S}$ of atoms (represented as atomary Prolog goals, possibly with templates) that we consider interesting enough to be monitored. They are all going to be annotated with the following two annotations, wrong/1 and missing/1. The annotating process is comfortable here because we do not have to write two sets of $|\mathcal{S}|$ isomorphic annotations, but only two parametric ones.

```
wrong(Q) :-
  exit Q => \+ wrong_instance_of(Q, _) else write('..WRONG').

% Q must be monotonic, see: var(X) vs. var(8)
wrong_instance_of(Q, Qu) :-
  is_unsatisfiable(Q, Qu),
  write('WRONG.. '), write(Qu), write(' -- unsatisfiable but provable:'), nl,
  nocheck, demo(Qu, Proof), check,
  write(' -- proof '), portray_clause(Proof).

missing(Q) :-
  ff Q within Parent => \+ missing_instance_of((Q within Parent), _)
    else write('..MISSING').

% Q must be monotonic, see: nonvar(X) vs. nonvar(8)
missing_instance_of((Q within Parent), Qv) :-
  is_valid(Q, Qv),
  write('MISSING.. '), write(Qv), write(' -- valid but not provable:'), nl,
  print_context(Parent, Qv),
  print_possible_matches(Qv).

is_unsatisfiable(Q, Qu) :- ...ask_user...
is_valid(Q, Qv) :- ...ask_user...
```

The annotations are rather self-explanatory, as they consist of reading (querying the oracle i.e. user), writing messages, and a call of the classic meta-predicate demo/2. To make it oracle-friendly, we add some bookkeeping (memoizing).

The basic idea is to monitor success and finite failure of our predicates.

If a goal Q succeeds with an answer substitution σ, Qσ may not have an unsatisfiable instance (*wrong atom*) in the intended semantics, given by the oracle. Similarly, if a goal finitely fails, it may not have a valid instance (*missing atom*) in the intended semantics. This presupposes, as usual for declarative degugging, that the predicates themselves are *also* declarative, in the sense of monotonicity (defined on page 1).

This *error monitoring* can be simple and helpful on its own, as illustrated in Sec. 4.6, but we can even have (heuristic) *error diagnosing* at no extra cost by observing that the first atom to violate one of the annotations doesn't depend on any other erroneous atoms of $\mathcal{S}$, and therefore gives a fair indication of the culprit of the error – modulo the predicates outside of $\mathcal{S}$, which are invisible.

An excerpt of execution should clarify this. The intended semantics, as elicited from the user, is shown boxed (the rows ending in '?' feed in the defaults shown in square brackets).

```
:- [ddebug]. % definition of wrong/1 and missing/2 annotations (see above)
:- wrong(isort(X,Y)), missing(isort(X,Y)).        % (A1)
:- wrong(insert(X,Y,Z)), missing(insert(X,Y,Z)).  % (A2)

% TheArt, program 17.13
isort([X|Xs], Ys) :- isort(Xs, Zs), insert(X, Zs, Ys).
isort([], []).

insert(X, [Y|Ys], [X,Y|Ys]) :- X >= Y. % bug
insert(X, [Y|Ys], [Y|Zs]) :- X > Y, insert(X, Ys, Zs).
insert(X, [], [X]).
```

The rows (A1) and (A2) set up a declarative debugger $\mathcal{D}(\mathcal{S}_0)$ for the isort program, with $\mathcal{S}_0 = \{$ insert(X,Y,Z), isort(X',Y') | for all X, Y, Z, X', Y' $\}$. After loading this piece of code, we can have the following debugging session.

```
| ?- isort([2,1], Result).
isort([],[])is valid [y]?
insert(1,[],[1])is valid [y]?
isort([1],[1])is valid [y]?
insert(2,[1],[2,1])is valid [y]? n
WRONG.. insert(2,[1],[2,1]) -- unsatisfiable but provable:
 -- proof insert(2, [1], [2,1]) :-
       (2>=1:-provable).
/* NOPE ..WRONG */
isort([2,1],[2,1])is valid [y]? n
WRONG.. isort([2,1],[2,1]) -- unsatisfiable but provable:
 -- proof isort([2,1], [2,1]) :-
       (isort([1],[1]):-(isort([],[]):-qed),(insert(1,[],[1]):-qed)),
       (insert(2,[1],[2,1]):-(2>=1:-provable)).
/* NOPE ..WRONG */
```

```
Result = [2,1] ? ;
insert(2,[],[2])is valid [y]?
insert(2,[1],[1,2])is valid [y]?
isort([2,1],[1,2])is valid [y]?

Result = [1,2] ? ;

no
| ?- bark(1). % enforces a Prolog interruption after the first warning

yes
| ?- isort([2,1,3], Result).
insert(3,[],[3])is valid [y]?
isort([3],[3])is valid [y]?
insert(1,[3],A)is unsatisfiable [y]? n
 -- your counterexample: insert(1,[3],A)= insert(1,[3],[1,3]).
MISSING.. insert(1,[3],[1,3]) -- valid but not provable:
 -- context isort([1,3], [1,3]) :-
        isort([3], A),
        insert(1, A, [1,3]).
 -- matching insert(1, [3], [1,3]) :-
        1>=3.
/* NOPE ..MISSING */
Prolog interruption (h for help)? a
Execution aborted
```

In case of missing answers we present also the calling context of the failed goal, to help further localize the error. This is made possible by the Context part of the annotations.

Observe that we did not specify any templates in our annotations, but we could have, in order to reduce the noise. That would correspond to shrinking the set $\mathcal{S}$. By incrementally adding/subtracting the atoms to/from $\mathcal{S}$ we can steer the precision and noise of the diagnosis. For example, if we replace the rows (A1) and (A2) with the following

```
:- wrong((isort(X,Y) with \+ sorted(Y))). % a calling template
:- missing((isort(X,Y) with \+ sorted(Y))).
:- wrong((insert(X,Y,Z) with Y \== [])).
:- missing((insert(X,Y,Z) with Y \== [])).
```

then we monitor the corresponding set $\mathcal{S}_1 \subset \mathcal{S}_0$. The atoms excluded from $\mathcal{S}_0$ are not bearing upon diagnosis, so this halves the amount of questioning by leaving out many boring bits, while still finding the culprit:

```
| ?- isort([2,1], Result).
insert(2,[1],[2,1])is valid [y]? n
WRONG.. insert(2,[1],[2,1]) -- unsatisfiable but provable:
```

```
 -- proof insert(2, [1], [2,1]) :-
        (2>=1:-provable).
/* NOPE ..WRONG */

Result = [2,1] ? ;
insert(2,[1],[1,2])is valid [y]?

Result = [1,2] ? ;

no
| ?- isort([2,1,3], Result).
insert(1,[3],A)is unsatisfiable [y]? n
 -- your counterexample: insert(1,[3],A)= insert(1,[3],[1,3]).
MISSING.. insert(1,[3],[1,3]) -- valid but not provable:
 -- context isort([1,3], [1,3]) :-
        isort([3], A),
        insert(1, A, [1,3]).
 -- matching insert(1, [3], [1,3]) :-
        1>=3.
/* NOPE ..MISSING */

no
```

To enhance the debugger to be able to handle goals with templates, all we had to do is add two lines of code, since the Goal part of our annotations is compositional in templates, i. e. in place of a goal Q we may substitute Q with T.

6 Related work (coda)

In Sec. 2 we started a comparison of debugging methods for Prolog, which we now round off with a few technical questions. For some more details and references about the declarative work in the area of Prolog verification, especially the annotation languages, see [7].

The idea of inserting checks at strategic places in a program Π ('instrumenting' it into a program Π') is a traditional debugging aid which is being continually refined upon. For C, the library macro *assert(·)* serves the purpose of aborting a program after issuing a message, in case the given expression is zero at the given point in the program.

Often it is tiresome to insert the checks manually. Automating this by a tool which instruments the program without the user's having to meddle with the sources is an attractive prospect. Such a tool has to be *non-interfering*, i. e. must not change the meaning of the program (the program must 'feel' the same).

One such kind of tool are tracers, doing no checking but just sampling the execution at some predefined points. A further step towards automated verification is coupling some actual checks with some points of execution. Traditionally pre/post conditions are being employed for this purpose (e. g. design by contract in Eiffel). In Prolog, one could observe several directions of refining this idea.

On acceptancy grounds, a smooth integration of the debugging tool in the Prolog mode of operation is recommended. Recall that Prolog operates as a read-interpret-write loop. So we can modify the Prolog reader, persuading it to read in not our program Π but the instrumented program Π' (this is what Nope does). Or we can even modify the program after it has been read in (this is what the original Advice does, hence its limitation on modifiable predicates). Or we can modify the Prolog interpreter, persuading it to compute with our program Π a bit differently, like it were Π'. This is what happens after trace or debug is issued, and also in the Advice re-implementation in Quintus Prolog, which we here refer to as AdviceQ, as well as in Opium.

The alternative to these 'smooth' methods is to write an own Prolog interpreter, i. e. a meta-interpreter, which is an efficiency problem. For practice, the program 'feeling the same' with and without the debugging tool means also, that the program shall not be slowed down too much. However, the main concern of preserving the program's meaning may not be compromised for efficiency.

Meta-interpretation won't be discussed here. Modifying Π is employed in Advice, Nope and CIAO. The more 'surgical' method of modifying Prolog is employed in the standard debugger, Opium and AdviceQ. The original Advice as well as Nope do not modify Prolog (we call them *non-invasive* upon Prolog). The standard debugger, AdviceQ and Opium do not modify Π.

In AdviceQ the advised predicates don't get modified at all, except calls to them get trapped by a mechanism that checks for and applies the relevant piece of advice at the various procedure box ports [3]. In Opium, all of the calls get trapped. The same low-level mechanism as used by the standard debugger. This accounts for the basic difference in runtime efficiency between the two paradigms, of Advice and Opium.

There is a technical issue to consider if a debugging tool is to be in daily use. For annotations which are currently un-checked, or for predicates without any annotations, the user expects that there be no perceptible decline in performance, so that the user doesn't have to abandon Π' in order to do 'some serious work' with Π. Recall the NDEBUG switch for *assert(·)*. This is fulfilled in Advice, AdviceQ and Nope, having idle overhead linear in the number of annotated predicates (e. g. in Nope this means three resolutions per annotated predicate).

One further criterion, discriminating between the transformation tools, is whether the transformation is performed on a goal basis or on the predicate basis. Advice or CIAO transform predicates, i. e. annotated predicates will be replaced by their 'instrumentations'. Nope transforms goals, at only the program points specified in the contexts. This further saves runtime overhead. The program points can be in the program or at top-level.

To conclude the discussion of software related to Nope, let us summarize the relationship to Advice. Advice [11] differs from Nope in several ways, the main being: it has unary assertions; uses unification instead of matching; has no contexts; negated occurences of a predicate cannot be checked on their own (as no built-ins can be checked). Regarding the first item, it is possible to enhance the goal replacement of Advice so that the calling pattern is being captured and made available to the advice checker. The second item is of course an advantage of Advice over Nope, since matching can be emulated by unification but not vice versa. The third and fourth item seem [3] to remain true restrictions of Advice.

Finally, let us summarize Nope. Nope is rather minimally invasive upon Π, and none upon Prolog, i. e. it won't modify the low-level of its host Prolog interpreter. Also, it is an (almost) standard Prolog program. The only essential facility outside of the ISO Prolog standard but necessary for Nope is term expansion [3]. The program transformation Nope is performing is goal replacement, at precisely the program points specified in the premisses. This is achieved by term expansion in a non-interfering way (in the sense of Lemma 3.12) and with small runtime overhead.

Acknowledgement

Many thanks for valuable comments upon previous drafts of this paper and other bits are due to C. Beierle, G. Meyer, R. Stärk, I. Đurđanović, M. Hermenegildo, U. Neumerkel, M. Carlsson, Vuk and the anonymous referees. This research was supported by the Deutsche Forschungsgemeinschaft (DFG), grant Be 1700/3-1.

References

[1] Edinburgh University A. I. Applications Institute. Edinburgh Prolog tools. ftp://sunsite.doc.ic.ac.uk/packages/prolog-pd-software/tools.tar.Z, 1988.

[2] Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.

[3] M. Carlsson. Personal communication, Sept. 1999.

[4] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[3] From [10]: "It is important to realise that a Prolog compiler that *only* supports vanillia Prolog without DCG and term_expansion/2 support is about as useful as a C compiler that lacks any preprocessing facilities and half its libraries."

[5] M. Ducassé. Opium: An extendable trace analyzer for Prolog. *J. of Logic Programming*, 39:177–223, 1999.

[6] M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.

[7] M. Kulaš. Annotations for Prolog – A concept and runtime handling. In *LOPSTR'99: 9th Internat. Workshop on Logic-based Program Synthesis and Transformation*. Universita' Ca' Foscari di Venezia, Rapporto di Ricerca CS-99-16, Sept. 1999. Also to appear in LNCS, Springer-Verlag, 2000.

[8] Gregor Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, FernUniversität Hagen, 1996. Also tutorial on ICLP'97. Source distribution http://www.fernuni-hagen.de/pi8/typical.

[9] U. Neumerkel. A programming course for declarative programming with Prolog. http://www.complang.tuwien.ac.at/ulrich/gupu/material/1997-gupu.ps.gz, 1997.

[10] R. A. O'Keefe. Re: parser in C. Thread in comp.lang.prolog, Sept. 1994.

[11] Richard A. O'Keefe. *advice.pl*. In DEC-10 Prolog Library [1]. Interlisp-like advice package, 1984.

[12] Quintus Corp., Palo Alto, CA. *Quintus Prolog Language and Library*, 1991. Release 3.1. Also library sources.

[13] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*. Edinburgh University Press, 1970.

[14] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[15] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/personal/staerk/lptp.html. Release 1.05.

[16] G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In *Proc. of the First Internat. Workshop on Automated and Algorithmic Debugging, Linköping, Sweden*, volume 749 of *LNCS*. Springer-Verlag, 1993.